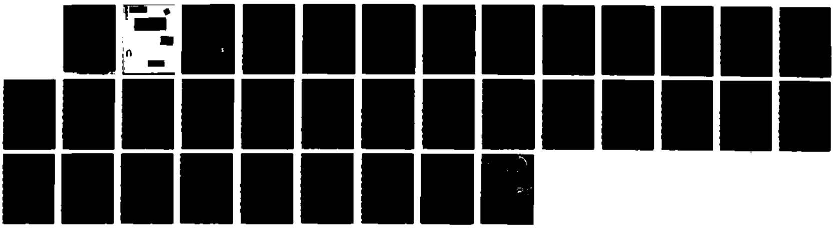
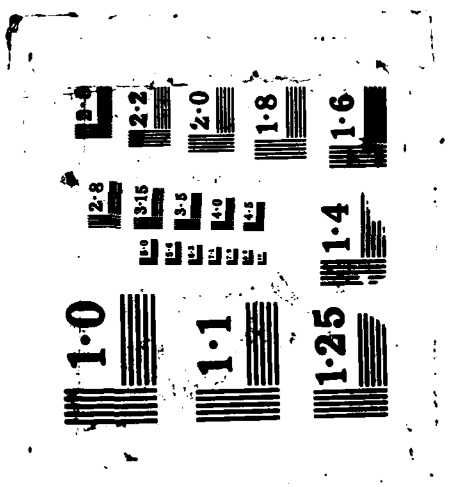
AD-A184 327 END OF YEAR REPORT FOR PARALLEL VISION ALGORITHM DESIGN 1/1
AND IMPLEMENTATIO (U) CARNEGIE-MELLON UNIV PITTSBURGH
PA ROBOTICS INST T KANADE ET AL JUN 87
UNCLASSIFIED CMU-RI-TR-87-15 DACA76-85-C-0002 F/G 12/5 NL




1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·8
3·15
3·5
4·0
4·5

End of Year Report for Parallel Vision Algorithm Design and Implementation

January 15, 1986–January 14, 1987

Takeo Kanade and Jon A. Webb

CMU-RI-TR-87-15

12

End of Year Report for Parallel Vision Algorithm Design and Implementation

January 15, 1986–January 14, 1987

Takeo Kanade and Jon A. Webb

CMU-RI-TR-87-15

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

June 1987

This research was supported by the Defense Advanced Research Projects Agency (DoD), monitored by the U.S. Army Engineer Topographic Laboratories under Contract DACA76-85-C-0002.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A184 327

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-RI-TR-87-15	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) End of Year Report for Parallel Vision Algorithm Design and Implementation		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Takeo Kanade and Jon A. Webb		8. CONTRACT OR GRANT NUMBER(s) DARPA (DoD) DACA76-85-C-0002
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie Mellon University The Robotics Institute Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE June 1987
		13. NUMBER OF PAGES 24
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) U.S. Army Engineer Topographic Laboratories		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The parallel vision algorithm design and implementation project was established to facilitate vision programming on parallel architectures, particularly low-level vision and robot vehicle control algorithms on the Carnegie Mellon Warp machine. To this end, we have (1) demonstrated the use of the Warp machine in several different algorithms; (2) developed a specialized programming language, called Apply, for low-level vision programming on parallel architectures in general, and Warp in particular; (3) used Warp as a research tool in vision, as opposed to using it only for research in parallel vision; (4) develped a significant library of low-level vision

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

87 9 8 065

(20 cont'd)

for use on Warp.

Table of Contents

List of Figures

Abstract

The parallel vision algorithm design and implementation project was established to facilitate vision programming on parallel architectures, particularly low-level vision and robot vehicle control algorithms on the Carnegie Mellon Warp machine. To this end, we have (1) demonstrated the use of the Warp machine in several different algorithms; (2) developed a specialized programming language, called Apply, for low-level vision programming on parallel architectures in general, and Warp in particular; (3) used Warp as a research tool in vision, as opposed to using it only for research in parallel vision; (4) developed a significant library of low-level vision programs for use on Warp.

1 Introduction

1986 was an exciting year of progress for the parallel vision algorithm design and implementation effort at Carnegie Mellon. We accomplished the following goals:

- We demonstrated Warp's use for road following, obstacle avoidance using stereo vision and ERIM laser range scanner data, MRI image processing, signal processing, and other vision algorithms.
- We developed a library of low-level vision algorithms, all written in the Warp programming language (W2). The library is based on the SPIDER FORTRAN subroutine library [32]. The current Warp vision library includes about 80 different Warp programs, covering edge detection, smoothing, image operations, Fourier transform, and so on. The actual number of routines in the SPIDER library covered by these Warp programs is about 100.
- We developed a specialized programming language called Apply for programming low-level vision algorithms on Warp. This programming language completely abstracts the underlying model of parallelism from the programmer. Moreover, Apply can generate efficient code for both Warp and Sun computers.
- We used Warp as a tool for research into vision algorithms, as opposed to purely being used as a tool for research into parallelism. This important step is rarely reached by many other advanced computers. The fact that we have already crossed it is evidence of Warp's good design for vision and a result of our efforts in making Warp accessible to vision researchers.

In the sections that follow, we discuss each of these accomplishments in detail. Section 3 discusses the Warp demonstration of August 29, 1986, in which we demonstrated a number of parallel vision algorithms, including robot road following, obstacle avoidance using both stereo and the ERIM laser range scanner, and Hough transform. Section 4 gives a summary of the current library status of the Spider subroutine library implementation. Section 5 gives a description of the Apply programming language and its implementation on Warp. Section 6 gives a brief description of some vision research being done on Warp.

2 Introduction to the Warp Machine

We describe the Warp architecture and then illustrate the Warp architectural decisions by comparisons with those made in a very different type of vision machine, namely bit-serial processor arrays.

2.1 The Warp Architecture and Programming Environment

This is a brief overview of Warp; more detail is available elsewhere [1, 4, 5, 10, 11, 13, 21, 22]. Warp has three components – the Warp processor array (*Warp array*), the interface unit (*IU*), and the *host*, as depicted in Figure 1. The Warp array performs the computation-intensive routines, for example, low-level vision routines. The IU handles the input/output between the array and the host, and generates addresses and control signals for the Warp array. The host executes the parts of the application programs that are not mapped onto the Warp array and supplies the data to and receives the results from the array.

The Warp array is a programmable, one-dimensional systolic array with identical cells called Warp cells. Data flows through the array on two data paths (X and Y), while addresses and systolic control signals travel on the Adr path (as shown in Figure 1).

Each Warp cell is implemented as a programmable horizontal microengine, with its own program memory and microsequencer. A Warp cell has a 32-bit wide data path, as depicted in Figure 2. The data path consists of two 32-bit floating-point processing elements: one multiplier and one ALU, a 4K-word memory for resident and temporary data, a 128-word queue for each communication channel, and a 32-word register file to buffer data for each floating-point unit. All these components are interconnected through a crossbar switch as shown in Figure 2.

The host consists of a VME-based workstation (currently a Sun 3/160), that serves as the master controller of the Warp machine, and an "external host," so named because it is *external* to the workstation. The workstation provides a UNIX environment for running application programs, and the external host provides a high data transfer

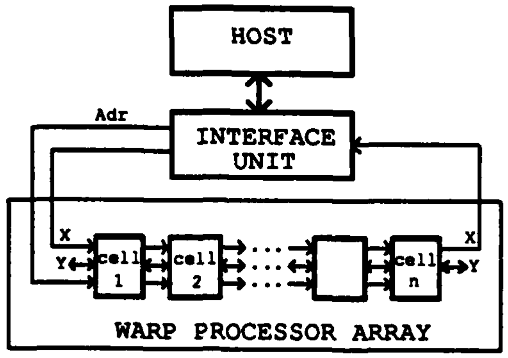


Figure 1: Warp machine overview

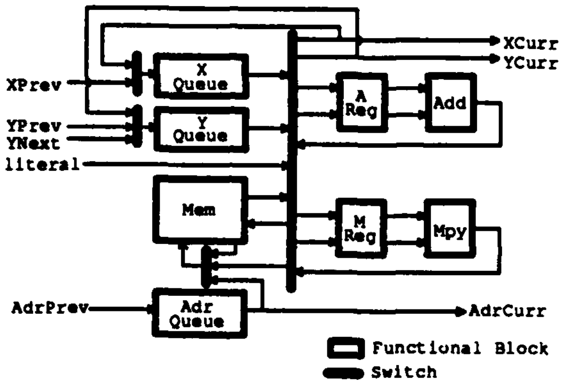


Figure 2: Warp cell data path

rate for communicating with the Warp array. The external host consists of three *stand-alone* 68020-based processors, which run without UNIX support to avoid operating system overheads. Two of the stand-alone processors are *cluster processors* responsible for transferring data to and from the Warp array, while the third one is a *support processor* that controls peripheral devices (e.g., the camera and the monitor), and handles interrupts originating in the clusters and Warp array.

The first two prototype machines used wire-wrap technology. The production version of Warp is implemented with PC boards; the 19" rack will be able to host the IU and up to 24 Warp cells. For the PC board version, each cell will have local data memory of 32K words, enlarged queues, and several other improvements [6].

The Warp programming environment is based on Common Lisp. A compiler, debugger, and execution environment are included. The programming language, W2, is approximately at the level of C or FORTRAN. Data structures such as arrays and scalars are included. Control structures include IF, WHILE, and FOR. The compiler hides from the programmer all the parallelism in the Warp machine except for the parallel execution of the Warp cells themselves. Communication between cells is implemented using SEND and RECEIVE, which transfer words between adjacent cells using an asynchronous protocol. On the prototype machines, the same program is executed on all cells; each cell has its own program counter, and can take different branches of conditionals, for example. This restriction is removed in the production machines. The debugger allows single stepping and source-level breakpoints, and allows the programmer to examine data structures within the Warp array. The execution environment manages the microcode and programs for the stand-alone processors, and aids the programmer in managing the memory of the external host.

Warp is integrated into the vision programming environment at Carnegie Mellon. Vision programming is based on the Generalized Image Library [14] which supports uniform access to images in files, frame buffers, memory, and printers. Presently, most vision programming is done in C/UNIX, using Suns and Vaxes; we expect to move into a Sun/Warp/Common Lisp based environment in the future.

2.2 Architectural Alternatives

We discuss the architectural decisions made in Warp by contrasting them with the decisions made in bit-serial processor arrays, such as the Connection Machine [34] and MPP [7]. We chose these architectures because they have also been used extensively for computer vision and image processing, and because the design choices in these architectures were made significantly different than in Warp. These differences help exhibit and clarify the design space for the Warp architecture.

We attempt to make our comparison quantitative by using benchmark data from a DARPA Image Understanding ("DARPA IU") workshop held in November 1986 to compare various computers for vision [29]. In this workshop, benchmarks for low and mid-level computer vision were defined and run by researchers closely associated with the computers being benchmarked on a wide variety of computers, including Warp and the Connection Machine [25].

We briefly review salient features of the Connection Machine, called CM-1, used in these benchmarks. It is a SIMD machine, consisting of an array of 64K bit-serial processing elements, each with 4K bits of memory. The processors are connected by two networks: one connects each processor to four adjacent processors, and the other is a 12-dimensional hypercube, connecting groups of 16 processors. The array is controlled by a host, which is a Symbolics 3640 Lisp machine. CM-1 is programmed in an extension to Common Lisp called *Lisp [24], in which references to data objects stored in the CM-1 array and objects on the host can be intermixed.

We should not compare benchmark performance on two different computers without mentioning two critical factors, namely cost and size. CM-1 is approximately one order of magnitude larger in volume and cost than Warp.

2.2.1 Programming model

Bit-serial processor arrays implement a *data parallelism* programming model, in which different processors process different elements of the dataset. In the Connection Machine, the programmer manipulates data objects stored in the Connection Machine array by the use of primitives in which the effect of a Lisp operator is distributed over a data object.

In systolic arrays, the systolic processors individually manipulate words of data. In Warp, we have implemented data parallelism programming models through the use of input and output partitioning. We have encapsulated input partitioning over images in a specialized language called Apply [15]. In addition to these models, the high interprocessor bandwidth of the systolic array allows efficient implementation of pipelining, in which not the data, but the algorithm is partitioned.

2.2.2 Processor I/O bandwidth and topology

Systolic arrays have high bandwidth between processors, which are organized in a simple topology. In the case of the Warp array, this is the simplest possible topology, namely a linear array. The interconnection networks in the Connection Machine allow flexible topology, but low bandwidth between communicating processors.

Bit-serial processing arrays may suffer from a serious bottleneck in I/O with the external world, because of the difficulty of feeding a large amount of data through a single simple processor. This bottleneck has been addressed in various ways. MPP uses a "staging memory" in which image data can be placed and distributed to the array along one dimension. The I/O bottleneck has been addressed by a new version of the Connection Machine, called CM-2 [33]. In this computer, a number of disk drives feed data into various points in the array. The CM-1 benchmark figures do not include image I/O: the processing is done on an image which has already been loaded into the array, and processing is completed with the image still in the array. Otherwise, image I/O would completely dominate processing time. For many purposes it is more convenient to process an image which is stored in a frame buffer or host memory, which is easier in Warp because of the high bandwidth between the Warp array and the Warp host. All the Warp benchmarks include I/O time from the host.

The high bandwidth connection between processors in Warp makes it possible for all processors to see all data in an image, while achieving useful image processing time. (In fact, because of the linear topology, there is no time advantage to limit the passage of an image through less than all processors). This is important in global image computations such as Hough transform, where any input can influence any output. For example, the transform of a 512×512 image into a 180×512 Hough space took 1.7 seconds on Warp, only 2.5 times as long as the 0.7 seconds for this computation on CM-1. The ratio here is far less than for a simple local computation on a large image, such as Laplacian and zero crossing.

The limited topology of the Warp array architecture has implications for some global algorithms, in which processing is done separately on different cells, then combined in a series of pairwise merge operations using a "divide and conquer" approach. For example, in the Warp border following algorithm for a 512×512 image, individual cells trace the borders of different portions of the image, then those borders are merged in a series of merge operations in the Warp array. The time for border following on Warp is 1100 milliseconds, significantly more than the 100 milliseconds the algorithm takes on CM-1.

2.2.3 Processor number and power

Warp has only ten parallel processing elements in its array, each of which is a powerful 10 MFLOPS computer. CM-1, on the other hand, has 64K processing elements, each of which is a simple bit-serial processor without floating-point capability. Thus, the two machines stand at opposite ends of the spectrum of processor power and number.

We find that the small number of processing elements in Warp makes it easier to get good use of the Warp array in problems where a complex global computation is performed on a moderate sized dataset. In these problems, not much data parallelism is "available." For example, the DARPA IU benchmarks included the computation of the two-dimensional convex hull [27] of a set of 1000 points. The CM-1 algorithm used a brush-fire expansion algorithm, which led to an execution time of 200 milliseconds for the complete computation. The same algorithm was implemented on Warp, and ran in 18 milliseconds. Similar ratios are found in the times for the minimal spanning tree of 1000 points (160 milliseconds on Warp versus 2.2 seconds on CM-1) and a triangle visibility problem for 1000 three dimensional triangles (400 milliseconds on Warp versus 1 second on CM-1).

Simple algorithms at the lowest level of vision, such as edge detection computations, run much faster on large arrays of processors such as the Connection Machine than Warp. This is because no communication is required between distant elements of the array, and the large array of processors can be readily mapped onto the large image array. For example, the computation of an 11×11 Laplacian [16] on a 512×512 image, followed by the detection of zero crossings, takes only 3 milliseconds on CM-1, as opposed to 400 milliseconds on Warp.

The floating-point processors in Warp aid the programmer in eliminating the need for low-level algorithmic analysis. For example, the Connection Machine used discrete fixed point approximation to several algorithms, including Voronoi diagram and convex hull. The use of floating-point made it unnecessary for the Warp programmer to make assumptions about data range and placement.

In conclusion, we see that bit-serial processor arrays excel in the lowest level of vision, such as edge detection. The CM-1's performance at this level exceeded Warp's by two orders of magnitude. However, specialized hardware must be used to eliminate an important I/O bottleneck to actually observe this performance. The use of the router in the Connection Machine allows it to do well also at higher levels of vision, such as border following. We also see that the more general class of programming models and use of floating-point hardware in Warp give it good actual performance in a wide range of algorithms, especially including complex global computations on moderate sized data sets.

3 Warp Demonstration

3.1 Introduction

This is a summary of the Warp demonstration of August 29, 1986, in which several different algorithms were demonstrated on Warp. The algorithms included several vision algorithms, as well as algorithms from signal processing and scientific computing.

3.2 Road-following

3.2.1 Task Description

The Terregator is controlled using algorithms running on Warp and the Sun to follow a road. The road is segmented from the background using shape and color. After each image is processed, the vehicle is steered so that it is in the center of the road by the time the next image is processed.

3.2.2 Algorithm Used

There are several steps in the algorithm. First, the image is reduced in size and classified into three regions using color. The colors used initially come from a training image where the road is outlined by the user using the cursor. Later they come from the previous image, as described below. The classification is done using a quadratic form, so that each color is defined as an ellipsoid in color space. Points are classified as left of road, road, or right of road depending on which is most like according to the measured means and covariance matrices of each feature.

In the second step, the image is cleaned up using simple image processing heuristics. The pixels classified as road are selected, and a binary grow/shrink algorithm is performed to remove isolated pixels and fill in cracks in the road. The image is then scanned row by row to determine the road position on each scan line.

In the third step, a Hough transform is performed to determine the actual road position given the estimated left and right road edges on each scan line. The Hough transform is constrained to find a road of the right width (using the width estimated from the training image).

In the fourth step the colors of the regions to the left and right of the road and the road itself are remeasured, giving improved estimates of the colors. The colors are remeasured for two reasons. First, the colors must be updated for the next image – the colors can change over time, due to lighting changes or scene changes, and we need to keep track of this. Second, it is possible for the colors to change radically from scene to scene, usually due to a video transmission error (loss of color is quite common with weak NTSC transmission) and so the first step sometimes completely fails – no road is detected using the old colors, or very little road is detected. In this case we take our best guess (assuming the road is straight ahead if we see nothing) and measure the current colors; this almost always allows us to recover from this kind of error.

In the fifth and sixth steps, the second and third steps are performed again, using the new color estimates.

In the seventh step, the vehicle is steered towards the center of the road, using the road position estimated by the Hough transform.

3.2.3 Mapping the Algorithm on Warp

The Sun, clusters, and Warp array all participate in this algorithm. The image reduction is performed by dividing the image into ten regions, and having each cell average and reduce one region. The color classification algorithm is performed on Warp; it is done by assigning each cell one tenth of the columns of the image, and broadcasting the color features to all cells. The grow and shrink operations are done using a similar method, except that each cell must overlap some of its columns with the next cell. The extraction of the road position on each scan line is done using the cluster processor. The Hough transform is performed on the Sun. The recalculation of the color features is done on Warp, again by splitting the columns of the image into ten parts, and giving each cell one tenth. The color features are then combined by each cell with the previous cell's.

3.2.4 Performance

The principal criterion in this algorithm is that the Terregator be driven successfully at full speed, 300 mm/sec (approx 1 km/hr), which is accomplished. Therefore, it was not necessary to optimize many of the steps.

The algorithm executes in 6 seconds/image. A comparable algorithm written by Richard Wallace took about 10 seconds/image on a SUN-3, but worked on a much coarser image: it was only 32×32 pixels. The Warp implementation makes only a small concession to image reduction: the image is 256×256, reduced from 512×512. Thus, the Warp algorithm is approximately 100 times faster than a SUN-3 (about a 200-fold improvement over a Vax 11/780).

If it were possible to run the Terregator faster, then the algorithm's running time could be decreased by about 2.25 seconds using two steps. First, cluster code that feeds the image directly into the Warp array instead of first storing the image in the cluster and then feeding the image from the cluster memory to Warp should be used. This will give approximately 0.75 seconds speedup. Second, the Hough transform can be implemented on Warp, giving approximately 1.5 seconds speedup. This speedup is obtained with no loss in image resolution.

Further speedup can be obtained by using a specially built device, currently under development at Carnegie Mellon, for feeding the image from the frame buffer to Warp (0.6 seconds) and further reduction of the image size. Approximately a factor of 3 is obtained for every reduction of 4 in image size, so a reduction to 128×128 images should give an algorithm that executes in 1 second.

3.3 Obstacle Avoidance with FIDO

3.3.1 Task Description

FIDO is a large program that is used to avoid obstacles. Two cameras are used to produce a three-dimensional estimate of possible obstacles positions. The program then plans and executes a path around any obstacles. Two Warp low level vision functions are used in the program. These are described below. Most of the rest is executed on the Sun workstation that runs Warp. One complete step currently takes 8 to 10 seconds.

3.3.2 Algorithm Used

Two algorithms are used: *Interest Operator*, and *Correlation*.

Interest Operator. Given a 256×256 pixel image, divided into 10×10 subimages, choose the most interesting points in each sub-image. A point is defined as interesting if it has a much different pixel value than any of its neighbors. For each point, calculate the intensity difference between that point and its neighbors above, to the right, to the upper right, and to the upper left. For each point, choose the minimum of these values. Then for each subimage, choose the maximum pixel difference of those left. This will give a set of 100 pixel positions and interest values.

Correlation. Given two 512×512 images and a set of 50 points corresponding to one of the images, find the corresponding points in the other image. To make matching more likely, two image pyramids are used. Each level of a pyramid is a lower resolution image obtained by averaging 4 pixels into 1. Matching is done between the two image pyramids starting with the lowest resolution level and continuing to higher resolution levels, with the match at the current level guiding where to search in the next level. Associated with each match is a correlation value, indicating how good the match is. The algorithm is as follows. The 4×4 pixel area is placed over the 8×8 pixel area in 5×5 different ways and the correlation of the overlapping pixels is calculated. The maximum correlation and its position are returned.

3.3.3 Mapping the Algorithm on Warp

Interest Operator. Each of the 10 cells in the array takes one tenth of the image (one tenth of the columns, all of the rows, with some overlap). Each cell then computes the "interestingness" of each point in its section, storing appropriate points and sending them to the host at the end of the function.

Correlation. Each cell gets 5 sets of areas and does a complete computation on each of them in turn.

3.3.4 Performance

Interest Operator. The interest operator function takes about 0.5 seconds on the Warp.

Correlation. The correlation takes about 1 second to execute. This will be shortened by moving some other operations to the Warp array.

3.4 Obstacle Avoidance Using ERIM Laser Range Scanner

3.4.1 Task Description

The ERIM laser range scanner is proving to be a highly effective device for obstacle avoidance. It provides direct three-dimensional information about the world, which can only be inferred using vision based techniques. The obstacle avoidance algorithm uses the ERIM scanner to construct a three-dimensional map of the world, which can then be navigated reliably. Three steps of the obstacle avoidance algorithm are implemented on Warp.

3.4.2 Algorithm Used

The first step of the algorithm takes the output from the ERIM scanner and calculates the three-dimensional coordinates of each point. This is a straightforward application of a polynomial formula to the value returned for each point.

The second step takes the three dimensional coordinates and constructs a grid of the world in front of the vehicle. The grid is divided into approximately 60 cells horizontally, and 64 cells vertically. A matrix is accumulated in each cell that will later be used to construct the covariance matrix of coordinates in each cell. The points are mapped into this grid, and the matrix is updated.

The third step solves for the eigenvalues of each covariance matrix, which gives the height and tilt of a best fit plane through the points in each grid cell.

The fourth step calculates the accessible and inaccessible regions of the grid using a connected-components analysis.

The fifth step calculates a path for the vehicle to take through the accessible regions of the grid.

3.4.3 Mapping the Algorithm on Warp

Only the first three steps are performed on Warp. In the first step of the algorithm the image is divided into ten regions, with each cell calculating the polynomial on one region. In the second step, it is the output array (the grid) that is divided ten ways, and each cell sees all of the input data. The third step computes the eigenvalues. This method was chosen because of the limited memory of the current Warp prototype.

3.4.4 Performance

Approximately five seconds are spent on the Sun in the first three steps of the algorithm. The first three steps take 350 ms on the Warp array, a speedup of five to ten. This allows us to achieve video rate for ERIM processing (500ms/frame). Just as significant as the execution time on Warp is the programming time. To obtain speed, the Sun algorithm uses several pre-computed lookup tables and is very obscure, while the power of the Warp makes it possible to code the algorithm in a straightforward manner, by simply doing the polynomial calculations as they are normally written.

The last steps of the algorithm do a single pass DP method to find a path through the obstacles, and then smooth the path by fitting arcs. There should be no problem in implementing the DP method in less than about 50 ms. Smoothing is more complex, and may be hard to implement on Warp. But in any case we can have the Sun do this planning while Warp is processing the next image. So the whole application should run in less than 500 ms, the ERIM frame rate, compared to 15 seconds (approximately) on the Sun – a factor of 30.

3.5 Path Planning

3.5.1 Task Description

Given a 512×512 image whose points are associated with various costs, the path planning task is to find the minimum-cost path from any point to a goal point. Path planning is often performed by an autonomous land vehicle (ALV).

3.5.2 Algorithm Used

The image is scanned in eight directions repeatedly. Every scan updates the current cost of each point, using a dynamic programming technique. The updated cost for a point is calculated from the most current costs of the neighboring points, including those costs just updated during the current scan. Scans terminate when no improvements are made on the cost of any point, for eight consecutive scans.

3.5.3 Mapping the Algorithm on Warp

For each scan the i^{th} cell is assigned to update points on the i^{th} row of the image; therefore 10 cells work on 10 rows simultaneously. Since costs for the $(i+1)^{st}$ row depend on those for the i^{th} row, the $(i+1)^{st}$ cell receives data from the i^{th} cell continuously. The Warp architecture supports this inter-cell communication efficiently.

3.5.4 Performance

Warp currently uses about 2 seconds to perform each scan. This time can be reduced to about 1 second by optimizing the program. Even without the optimization, Warp is already about 60 times faster than a VAX 11/780 for this path planning problem.

3.6 ALV Algorithms: Finding Lines by Hough Transform

3.6.1 Task Description

The task is to detect lines in a noisy 512×512 binary image. Assume that the origin *(0,0)* is at the lower left-hand corner of the image, with the x-axis along the bottom row. The output is a 180×512 array of nonnegative integers constructed as follows: for each pixel *(x,y)* having value 1 in the input image, and each *i*, $0 <= i < 180$, add 1 to the output in position *(i,j)*, where *j* is the perpendicular distance (rounded to the nearest integer) from *(0,0)* to the line through *(x,y)* making an angle of *i* degrees with the x-axis (measured counterclockwise). (If the input image has many collinear 1's, they will produce a high-valued peak in the output.)

3.6.2 Algorithm Used

The image is scanned in row order. For each pixel having the value 1, and for each angle *i* the following steps are carried out:

1. Distance to the origin from a line through the pixel and making *i* degrees with the x-axis is computed.
2. Distance is rounded to the nearest integer which gives *j*.
3. Output array is incremented at position *(i,j)*.

3.6.3 Mapping the Algorithm on Warp

Each cell gets 1/10th of the set of angles. Each cell gets every pixel, does the calculations for its subset of angles and generates 1/10th of the resulting image which is output at the end. The algorithm was implemented on Warp with a set of 50 angles (step of 3.6 degrees). Searching for peaks in the 50×512 output is done on the host and for each maximum found in the output image at position (i,j) a line with distance j from the origin and making an angle i degrees with the x-axis is drawn.

3.6.4 Performance

For a 512×512 input image and 50 angles, the Warp processing time is about 2 seconds. Each cell does about 6 million floating-point operations per pixel which gives a total of 30 MFLOPS for the Warp array. Warp takes 2 seconds to execute the algorithm. Carefully written code on a Vax 11/780 runs 390 times slower.

3.7 Graph Algorithms: Finding Minimum-cost Path

3.7.1 Task Description

A random graph of 350 nodes and an arbitrary number of edges between nodes are given. Each edge is assigned a cost. The cost of a path is defined to be the sum of the costs of the edges in the path. The minimum cost path task is to find the minimum-cost path between all pairs of nodes.

3.7.2 Algorithm Used

Wallshall's algorithm is used [3]. The kernel of the algorithm is given below:

```
for ( k = 0 ; k < 350 ; k ++) {
  for ( i = 0 ; i < 350 ; i++) {
   for ( j = 0 ; j < 350 ; j++) {
      C[i][j] = MIN( C[i][j] ,
                    C[i][k]+C[k][j]) ;
}}}
```

3.7.3 Mapping the Algorithm on Warp

Each cell performs one outmost iteration (indexed by variable k) of the above algorithm. Each run of the Warp array performs 10 iterations of the above algorithm. To complete the task, 35 Warp runs are necessary.

3.7.4 Performance

Warp currently uses about 16 seconds to complete the whole task, including the overhead of the host control. Warp is about 98 times faster than the VAX 11/780 for this problem.

3.8 Scientific Computing: Solving Elliptic PDEs with SOR

3.8.1 Task Description

SOR is used to solve a boundary value problem (Dirichlet type) of elliptic partial differential equations on a square region $[0,1]\times[0,1]$.

$$(P(x,y)\times U_x)_x+(Q(x,y)\times U_y)_y+K(x,y)\times U=F(x,y),$$

where U is the unknown and P, Q, K, F are given functions. The region is discretized into a mesh of 225×225 and a staggered grid difference approximation is applied to derive a linear system of equations with 50,625 unknowns. The task is to solve this linear system of equations.

For this demonstration, the model problem of a Poisson equation is selected as a benchmark. That is $U_{xx}+U_{yy}=0$, the left boundary and right boundary of the square region are set to 0, the upper boundary and the lower boundary are set to non-zero given functions.

3.8.2 Algorithm Used

The Successive Over-Relaxation (SOR) iterative method is used for solving the PDE problem. The optimal relaxation coefficient $\omega = 1.972$ is used to accelerate the convergence.

3.8.3 Mapping the Algorithm on Warp

Each warp cell performs one iteration of SOR. The unknown vector U is sent through the Warp array in raster order, as it is done in sequential algorithms. Once a cell completes the update of one row of mesh points, and sends them to the next cell, the cell updates the next row while the next cell starts updating the first row for the next iteration, and so on for the whole array of 10 cells.

In one run through the Warp cell, 10 iterations of SOR is done with the same relaxation coefficient ω. The convergence test is then performed on the host. If the solution has not converged yet, another run of Warp is carried out. For one given benchmark example, the algorithm takes 322 iterations to reach the stopping criterion of relative error 1.0e-4, which means 33 runs of Warp array.

3.8.4 Performance

Warp currently takes about 16 seconds to complete a solution which requires 330 iterations, including the host overhead and convergence test. Warp is about 440 times faster than a VAX 11/780 for the SOR algorithm.

3.9 Scientific Computing: Adaptive Beamforming for Sonar Using SVD

3.9.1 Task Description

Adaptive beamforming is a method by which dynamic weighting may be applied to a linear array of receiving elements, such as sonar hydrophones. The phase weightings that are applied alter the overall array beam pattern in such a way as to maximize the gain in one desired direction while minimizing it in the others with respect to some particular criteria.

This beam steering effect is used in this demonstration, to reduce the gain of the array in the direction of some large interference signals while improving it in the direction of weak sources.

We perform the demonstration using synthesised frequency domain data, i.e., commencing the algorithm *after* the FFT operation. Additionally, instead of performing the inner product of the weight vector with the frequency domain matrix, we plot a function of the weight vector in a polar format as this shows the adaptation of the array beamshape in the face of interfering sources. In particular, we display the actual beampattern both before adaptation (i.e., zero weighting) and during adaptation, showing that performace improves as more terms are added to the weighting function. The beam pattern is shown in polar coordinates, from 0 to 90 degrees. The angles of interference are also shown.

3.9.2 Algorithm Used

Cornerstone of the demonstration, from an algorithm point of view, is the execution of a complex Singular Value Decomposition (SVD). We perform the complex SVD via a 100×100 real SVD. The SVD of a real matrix A is defined as:

$$U^T A V = \Sigma,$$

where Σ is a $p \times p$ nonnegative diagonal matrix, and U and V are $m \times m$ and $n \times n$ orthogonal matrices, respectively. The nonzero elements in the diagonal of Σ are the singular values of A. The algorithm is based on the Hestenes methods. The method generates a sequence of J's such that:

$$AJ_1 \ldots J_\sigma = U\Sigma.$$

Each J_j is obtained by a set of plane rotations that orthogonalize columns. We use Schimmel and Luk's ordering [30].

We have a set of *rotations* applied to A and each complete rotation set is called a *sweep*. The algorithm follows two distinct phases:

1. A rotation set is carried out. It consists of inner products between "adjacent" columns. For each column-pair three values are generated. For an $n \times n$ problem a total of $3 \times n/2$ (or $3 \times (n/2-1)$, depending on whether the rotation is odd or even) values are generated.
2. From these $n/2$ triplets, the sine and cosine of the rotation angle are calculated. These values make up the structure of the tridiagonal $\mathbf{J}_j$ matrix, which is then multiplied by **A** to generate the new **A** matrix. A new rotation is then carried out.

A sweep consists of n of the above rotation sets (either "odd" or "even"). When the sweep is completed, a convergence criteria can be applied. Alternatively, it is known that $\log(n)$ sweeps are sufficient to compute the singular values.

3.9.3 Mapping the Algorithm on Warp

A 100×100 problem is considered. The algorithm has been mapped on Warp as follows. First, the A matrix is stored into Warp. In particular, each Warp cell contains ten *rows*.

When a rotation set is executed, each cell starts computing the inner products of the fifty sections of column-pairs it contains. Each cell computes the inner product of each column in a column pair individually, as well as the inner product of the two columns together, producing a triplet of numbers. The first cell then passes the partial results to the next cell, which adds them to those it generated, and passes them along to the next cell. The end result is a stream of triplets being sent to one of the cluster processors.

The cluster processor receiving the triplets computes the sine and cosine, and sends them to the other cluster processor. Simulation has shown that, to reliably converge, it is necessary to carry out this computation in double-precision. Luckily, the cluster processors feature a 68881 floating-point co-processor, which has double-precision floating-point computation capability. Simulation has also shown that double precision is not needed in other phases of the computation. This was very important for us because the Warp array works on single-precision only. Finally, the computation on the cluster requires both full-precision division and square-root, none of which is available as a hardware primitive on the Warp array.

When the second cluster starts receiving the sines and cosines, it sends them back to the Warp array, which performs matrix multiplication (i.e., $\mathbf{AJ}_j$). Then, a new rotation is started. No convergence criteria is implemented, but, rather, $\log(n)$ sweeps are carried out.

The correctness of the algorithm has been checked with the SVD EISPACK routine. Both Warp and the EISPACK routines started from the same matrix, generated with the EISPACK random number generator routine.

3.9.4 Performance

The performance of the whole adaptive beamforming application is strictly dependent on the speed of execution of the SVD. The following numbers are for a 100×100 problem. The array executes a sweep in 150ms., which is about 40 MFLOPS. A total of 8 sweeps are performed. A 100×100 problem, defined as "A matrix in core – singular values in core," takes about 6 seconds on Warp. The EISPACK routine on the same matrix took about 23 seconds on a Vax 8650. Therefore, Warp is about 23 times faster than a Vax 11/780. This factor of 23 is confirmed when the the time for the whole adaptive beamforming algorithm is measured.

Further speed-up can be obtained by assigning the computation of the rotation parameters to *both* cluster processors. An improvement of about two seconds is expected. The execution time on the current Warp of a 100×100 SVD problem cannot go below 4 second, therefore. Further improvements require a dedicated "Boundary Processor" – under development at Carnegie Mellon – to compute the rotation parameters. With such a component it is reasonable to assume that Warp could execute the task in less than two seconds.

3.10 Signal Processing: 2-D Correlation Using FFT

3.10.1 Task Description

A discrete 512×512 two-dimensional cyclic correlation of two input images is performed. The ouput image should have a maximum where the input images match.

3.10.2 Algorithm Used

The correlation is done with a 512×512 Fast Fourier Transform (FFT). First, the direct FFT's of the two input images are computed. Then, the conjugate of the direct FFT of one of the images is multiplied by the direct FFT of the other, and the inverse FFT of the result of this multiplication is computed and displayed.

3.10.3 Mapping the Algorithm on Warp

This is basically an application of the FFT algorithm already implemented on Warp. The algorithm implemented on Warp is a systolic 9-stage constant geometry 512-point one-dimensional FFT where each cell does one stage, that is, takes the partial results from the previous cell, does the calculations and passes the new partial results to the next cell. A 512×512 2-D (direct or inverse) FFT is done by calculating the 512-point 1-D FFT's of the 512 rows in a first pass and then the 512-point 1-D FFT's of the 512 columns of the result of the first pass. That gives a total of 1024, 512-point 1-D FFT's.

3.10.4 Performance

The Warp processing time for a (direct or inverse) 512×512 2-D FFT is about 0.3 sec and the actual processing time (considering I/O overhead) is about 2.5 sec which is about 40 times faster than FACOM M-160AD. For this correlation application, where there are also other computations being done on Warp like float-to-integer conversions for displaying and complex multiply, the total time is about 8 seconds. This implies that Warp is about 300 times faster than the Vax 11/780. The speed-up can be substantially increased when the host code for the application is improved.

3.11 Mandelbrot Sets

3.11.1 Task Description

This demonstrates the application of Warp in mathematics. A newly active field of mathematics is problem solving using computer graphics. The idea is to write a program that will produce results that can be *displayed*, rather than just printing a meaningless sequence of numbers.

3.11.2 Algorithm Used

The Mandelbrot and Julia Sets are derived from simple functions that are computed iteratively. The basic function for these sets is

$$X_{i+1}=X_i \times X_i + c$$

where X_0 and c are determined at the start of the repetition. What makes the computation interesting is that both X_0 and c are complex numbers. Because of this, the simple operation above produces very complicated results. By picking a group of starting c or X_0 values, images can be created. Mandelbrot and Julia Sets are fractal curves.

3.11.3 Mapping the Algorithm on Warp

Each Warp cell works on 1/10th of the image, performing the same operations as its neighbors but on different input parameters.

3.11.4 Performance

The program that runs on the Warp does a 512×512 image with 256 iterations (each iteration is 10 floating point operations) in 7.5 seconds. An equivalent program takes 12 minutes on a Vax 11/780. Such curves can only be generated on computers with floating point capabilities.

3.12 Summary of Measured Warp Speed-ups

Some of the tasks presented in this document have been implemented on various computers. In particular, Vax 8650, Vax 11/780 with floating-point accelerator, and SUN-3 with 68881 floating-point co-processor have been used. We have normalized the performance of Warp to that of a Vax 11/780 by using the following, approximate factors:

SUN-3 is approximately 2 times faster than Vax 11/780, and
Vax 8650 is approximately 6 times faster than Vax 11/780.

The label N/A means that a meaningful comparison does not apply. The mark (*) means that the speed-up can be further improved by a factor of at least two, when the application program for Warp is optimized. Explanations are given in the section where the task is discussed.

Task	Speed-up over Vax 11/780	
Road-following	200	
Obstacle Avoidance with FIDO	N/A	
ALV Algorithms: Obstacle Avoidance Using ERIM Laser Range Scanner	60	(*)
ALV Algorithms: Path Planning	60	(*)
ALV Algorithms: Finding Lines by Hough Transform	390	
Graph Algorithms: Finding Minimum-cost Path	98	
Scientific Computing: Solving Elliptic PDEs with SOR	440	
Scientific Computing: Adaptive Beamforming for Sonar using SVD	23	(*)
Signal Processing: 2-D Correlation Using FFT	300	(*)
Mandelbrot Sets	95	

4 Vision Library Implementation Status

As an imporant aid for the Warp programmer, and to facilitate use of Warp by people who do not want to program Warp, we have created a library of low-level vision routines. All of these routines are written in the Warp programming language (W2)–earlier implementations of some routines in Warp microcode (W1 and W0) which were superseded by W2 code. In the future, we plan to rewrite some of the programs in the Apply language, which will also give these routines the capability of being run efficiently on computers other than Warp, such as the Sun. The library is based on the SPIDER FORTRAN subroutine library [32]. The current Warp vision library includes about 80 different Warp programs, covering edge detection, smoothing, image operations, Fourier transform, and so on. The actual number of routines in the SPIDER library covered by these Warp programs is about 100.

5 Low Level Vision on Warp and the Apply Programming Model

5.1 Introduction

In computer vision, the first, and often most time-consuming step in image processing is *image to image operations*. In this step, an input image is mapped into an output image through some local operation that applies to a window around each pixel of the input image. Algorithms that fall into this class include: edge detection, smoothing, convolutions in general, contrast enhancement, color transformations, and thresholding. Collectively, we call these operations low-level vision. Low-level vision is often time consuming simply because images are quite large – a typical size is 512×512 pixels, so the operation must be applied 262,144 times.

Fortunately, this step in image processing is easy to speed up through the use of parallelism. The operation applied at every point in the image is often independent from point to point, and also does not vary much in execution time at different points in the image. This is because at this stage of image processing, nothing has been done to differentiate one area of the image from another, so that all areas are processed in the same way. Because of these two characteristics, many parallel computers achieve good efficiency in these algorithms, through the use of *input partitioning* [23], also known as data parallelism.

We discuss a particular parallel computer, the Warp machine, which has been developed for image and signal processing, and describe its use at this level of vision. We also define a language, *Apply*, which is specifically designed for implementing these algorithms. Apply runs on the Warp machine, and in C under UNIX, with good efficiency in both cases. Therefore, the programmer is not limited to developing his programs just on Warp, although they run much faster (typically 100 times faster) there; he can do some development under the more generally available UNIX system.

We consider Apply and its implementation on Warp to be a significant development for image processing on supercomputers in general. The programmer of a supercomputer usually makes a substantial commitment to the particular supercomputer he is using because he cannot expect that his code will run efficiently on any other computer. This limits the use of supercomputers, because such a great investment in coding is required that only truly committed users will make this investment. With Apply however, the programmer can recompile his code for other machines. Right now, only UNIX systems and Warp run Apply programs. But since we include a definition of Apply as it runs on Warp, and because most parallel computers support input partitioning, it should be possible to implement it on other supercomputers as well. Once this is done, the Apply programmer will be able to port his code easily to many different computers, lengthening the lifetime of his code and lessening the commitment he must make to a particular computer.

Apply also has implications for benchmarking of new image processing supercomputers. Currently, it is hard to compare these computers, because they all run different, incompatible languages and operating systems, so the same program cannot be tested on different computers. Once Apply is implemented on different supercomputers, it will be possible to test their performance on an important class of image operations, namely low-level vision.

Apply is not a panacea for these problems; it is an application-specific language, which is potentially machine independent. It cannot be used for all vision algorithms, and even some low-level vision algorithms cannot be efficiently expressed in it as it is currently defined.

We begin by discussing our early work on low-level vision, where we developed the input partitioning method on Warp. Then we define and discuss Apply. Following this, we describe how Apply might be implemented on other computers.

5.2 Low-level vision on Warp

We map low-level vision algorithms onto Warp by the *input partitioning* method. On a Warp array of ten cells, the image is divided into ten regions, by column, as shown in Figure 3. This gives each cell a tall, narrow region to process; for 512×512 image processing, the region size is 52 columns by 512 rows. To use technical terms from weaving, the Warp cells are the "warp" of the processing; the "weft" is the rows of the image as it passes through the Warp array.

The image is divided in this way using a series of macros called `GETROW`, `PUTROW`, and `COMPUTEROW`.

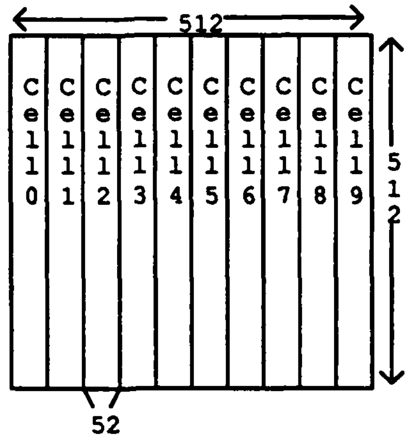


Figure 3: Input partitioning method on Warp

GETROW generates code that takes a row of an image from the external host, and distributes one-tenth of it to each of ten cells. The programmer includes a GETROW macro at the point in his program where he wants to obtain a row of the image; after the execution of the macro, a buffer in the internal cell memory has the data from the image row.

The GETROW macro works as follows. The external host sends in the image rows as a packed array of bytes – for a 512-byte wide image, this array consists of 128 32-bit words. These words are unpacked and converted to floating point numbers in the interface unit. The 512 32-bit floating point numbers resulting from this operation are fed in sequence to the first cell of the Warp array. This cell takes one-tenth of the numbers, removing them from the stream, and passes through the rest to the next cell. The first cell then adds a number of zeroes to replace the data it has removed, so that the number of data received and sent are equal.

This process is repeated in each cell. In this way, each cell obtains one-tenth of the data from a row of the image. As the program is executed, and the process is repeated for all rows of the image, each cell sees an adjacent set of columns of the image, as shown in Figure 3.

We have omitted certain details of GETROW – for example, usually the image row size is not an exact multiple of ten. In this case, the GETROW macro pads the row equally on both sides by having the interface unit generate an appropriate number of zeroes on either side of the image row. Also, usually the area of the image each cell must see to generate its outputs overlaps with the next cell's area. In this case, the cell copies some of the data it receives to the next cell. All this code is automatically generated by GETROW.

PUTROW, the corresponding macro for output, takes a buffer of one-tenth of the row length from each cell and combines them by concatenation. The output row starts as a buffer of 512 zeroes generated by the interface unit. The first cell discards the first one-tenth of these and adds its own data to the end. The second cell does the same, adding its data after the first. When the buffer leaves the last cell, all the zeroes have been discarded and the first cell's data has reached the beginning of the buffer. The interface unit then converts the floating point numbers in the buffer to zeroes and outputs it to the external host, which receives an array of 512 bytes packed into 128 32-bit words. As with GETROW, PUTROW handles image buffers that are not multiples of ten, this time by discarding data on both sides of the buffer before the buffer is sent to the interface unit by the last cell.

During GETROW, no computation is performed; the same applies to PUTROW. Warp's horizontal microword, however, allows input, computation, and output at the same time. COMPUTEROW implements this. Ignoring the complications mentioned above, COMPUTEROW consists of three loops. In the first loop, the data for the cell is read into a memory buffer from the previous cell, as in GETROW, and at the same time the first one-tenth of the output buffer is discarded, as in PUTROW. In the second loop, nine-tenths of the input row is passed through to the next cell, as in GETROW; at the same time, nine-tenths of the output buffer is passed through, as in PUTROW. This loop is unwound by COMPUTEROW so that for every 9 inputs and outputs passed through, one output of this cell is computed. In the third loop, the outputs computed in the second loop are passed on to the next cell, as in PUTROW.

There are several advantages to this approach to input partitioning:

- Work on the external host is kept to a minimum. In the Warp machine, the external host tends to be a bottleneck in many algorithms; in the prototype machines, the external host's actual data rate to the array is only about 1/4th of the maximum rate the Warp machine can handle, even if the interface unit unpacks data as it arrives. Using this input partitioning model, the external host need not unpack and repack bytes, which it would have to if the data was requested in another order.
- Each cell sees a connected set of columns of the image, which are one-tenth of the total columns in a row. Processing adjacent columns is an advantage since many vision algorithms (e.g., median filter [18]) can use the result from a previous set of columns to speed up the computation at the next set of columns to the right.
- Memory requirements at a cell are minimized, since each cell must store only 1/10th of a row. This is important in the prototype Warp machines, since they have only 4K words memory on each cell.
- The image is processed in raster order, which has for a long time been a popular order for accessing data in an image. This means that many efficient algorithms, which have been developed for raster-order image processing, can be used.
- An unexpected side effect of this I/O model was that it made it easier to debug the hardware in the Warp machine. If some portion of a Warp cell is not working, but the communication and microsequencing portions are, then the output from a given cell will be wrong, but it will keep its proper position in the image. This means that the error will be extremely evident – typically a black stripe is generated in the corresponding position in the image. It is quite easy to infer from such an image which cell is broken!

5.3 Introduction to Apply

The Apply programming model is a special-purpose programming approach which simplifies the programming task by making explicit the parallelism of low-level vision algorithms. We have developed a special-purpose programming language called the Apply language which embodies this parallel programming approach. When using the Apply language, the programmer writes a procedure which defines the operation to be applied at a particular pixel location. The procedure conforms to the following program model:

- It accepts a window or a pixel from each input image.
- It performs arbitrary computation, usually without side-effects.
- It returns a pixel value for each output image.

The Apply compiler converts the simple procedure into an implementation which can be run efficiently on the Warp supercomputer, or on a uni-processor machine in C under UNIX.

5.3.1 The Apply Language

The Apply language is designed for programming image to image computations where the pixels of the output images can be computed from corresponding rectangular windows of the input images. The essential feature of the language is that each operation is written as a procedure for a single pixel position. The Apply compiler generates a program which executes the procedure over an entire image. No ordering constraints are provided for in the language, allowing the compiler complete freedom in dividing the computation among processors. As a consequence of this, however, Apply does not allow the output of the computation at one pixel location to be used as the input for the same computation at a nearby pixel, as is done in several low-level vision operators, such as uniform smoothing (see FLWL2 [32]) or median filter [18]. Provision for limited feedback is an area for future research.

Each procedure has a parameter list containing parameters of any of the following types: *in*, *out* or *constant*. Input parameters are either scalar variables or two-dimensional arrays. A scalar input variable represents the pixel value of an input image at the current processing coordinates. A two-dimensional array input variable represents a window of an input image. Element (0,0) of the array corresponds to the current processing coordinates.

Output parameters are scalar variables. Each output variable represents the pixel value of an output image. The final value of an output variable is stored in the output image at the current processing coordinates.

Constant parameters may be scalars, vectors or two-dimensional arrays. They represent precomputed constants which are made available for use by the procedure. For example, a convolution program would use a constant array for the convolution mask.

The reserved variables `ROW` and `COL` are defined to contain the image coordinates of the current processing location. This is useful for algorithms which are dependent in a limited way on the image coordinates.

Figure 4 is a grammar of the Apply language. The syntax of Apply is based on Ada [2]; we chose this syntax because it is familiar and adequate, and because we do not wish to create yet another new language syntax, nor do we consider language syntax to be an interesting research issue. However, as should be clear, the application dependence of Apply means that it is not an Ada subset, nor is it intended to evolve into such a subset.

Apply is strongly typed and does not allow assignment of integer expressions to floating variables or floating expressions to integer variables. Mixed expressions are also disallowed. An integer expression may be explicitly converted to floating by means of the pseudo-function `FLOAT` and a floating expression can be converted to integer by using the pseudo-function `INTEGER`.

```
procedure               ::=   PROCEDURE function-name ( function-args )
                              IS
                                   variable-declarations
                              BEGIN
                                   statements
                              END function-name;

function-args           ::=   function-argument , function-args
                        |     function-argument

function-argument       ::=   var-list : parameter-source type
                        |     var-list : IN type BORDER const-expr

var-list                ::=   variable , var-list
                        |     variable

parameter-source        ::=   IN
                        |     OUT
                        |     CONST

variable-declarations
                        ::=   var-list : type ; variable-declarations
                        |     Empty

type                    ::=   ARRAY ( dimension-list ) OF elementary-type
                        |     elementary-type

dimension-list          ::=   range , dimension-list
                        |     range

range                   ::=   int-expr .. int-expr

elementary-type         ::=   sign object
                        |     object

sign                    ::=   SIGNED
                        |     UNSIGNED
```

```
object              ::=  BYTE
                    |    INTEGER
                    |    FLOAT

statements          ::=  statement ; statements
                    |    statement ;

statement           ::=  assignment-stmt
                    |    if-stmt
                    |    for-stmt

assignment-stmt     ::=  scalar-var := expr

scalar-var          ::=  variable
                    |    variable ( subscript-list )

subscript-list      ::=  int-expr , subscript-list
                    |    int-expr

expr                ::=  expr + expr
                    |    expr - expr
                    |    expr * expr
                    |    expr / expr
                    |    ( expr )
                    |    pseudo-function ( expr )

if-stmt             ::=  IF bool-expr THEN
                             statements
                         END IF
                    |    IF bool-expr THEN
                             statements
                         ELSE
                             statements
                         END IF

bool-expr           ::=  expr < expr
                    |    expr <= expr
                    |    expr = expr
                    |    expr >= expr
                    |    expr > expr
                    |    expr /= expr

for-stmt            ::=  FOR int-var IN range LOOP
                             statements
                         END LOOP
```

Figure 4: Grammar of the Apply language

Variable names are alpha-numeric strings of arbitrary length, commencing with an alphabetic character. Case is not significant, except in the preprocessing stage which is implemented by the *m4* macro processor [20].

Parameter variables refer to images, so they can be only one or two dimensional; function variables can be of any dimension. Both the C and FORTRAN forms of array indexing (with brackets or commas separating dimensions) are allowed. `BYTE`, `INTEGER`, and `FLOAT` refer to (at least) 8-bit integers, 16-bit integers, and 32-bit floating point numbers. `BYTE` values are converted implicitly to `INTEGER` within computations. The actual size of the type may be larger, at the discretion of the implementor.

Some restrictions in the current implementation of Apply result from limitations in the W2 compiler [12] for the prototype Warp machine.

- There are no Boolean *and*, *or*, or *not* operations.
- There may not be any *for* loops inside of *if* statements.
- *For* loops must have constant integer lower and upper bounds.
- There are no structured variables, only scalar variables and arrays.
- There is no facility for writing functions which invoke other functions.

We expect these limitations will be lifted in the future, once Apply is implemented on the printed-circuit board version of Warp.

5.3.2 An Implementation of Sobel Edge Detection

As a simple example of the use of Apply, let us consider the implementation of Sobel edge detection. Sobel edge detection is performed by convolving the input image with two 3 by 3 masks. The horizontal mask measures the gradient of horizontal edges, and the vertical mask measures the gradient of vertical edges. Diagonal edges produce some response from each mask, allowing the edge orientation and strength to be measured for all edges. Both masks are shown in Figure 5.

```
| 1  2  1 |        | 1  0 -1 |
| 0  0  0 |        | 2  0 -2 |
| -1 -2 -1 |       | 1  0 -1 |
```

Horizontal *Vertical*

Figure 5: The Sobel convolution masks

An Apply implementation of Sobel edge detection is shown in Figure 6. The lines have been numbered for the purposes of explanation, using the comment convention. Line numbers are not a part of the language.

```
procedure sobel (inimg  : in array (-1..1, -1..1)        -- 1
                          of byte
                          border 0,
                 thresh : const float,
                 mag    : out float)
is                                                       -- 2
    horiz, vert : integer;                               -- 3
begin                                                    -- 4
    horiz := inimg(-1,-1) + 2 * inimg(-1,0)              -- 5
        + inimg(-1,1) - inimg(1,-1)
        - 2 * inimg(1,0) - inimg(1,1);
    vert := inimg(-1,-1) + 2 * inimg(0,-1)               -- 6
        + inimg(1,-1) - inimg(-1,1)
        - 2 * inimg(0,1) - inimg(1,1);
    mag := sqrt(FLOAT(horiz)*FLOAT(horiz)                -- 7
              + FLOAT(vert)*FLOAT(vert));
    if mag < thresh then                                 -- 8
        mag := 0.0;                                      -- 9
    end if;                                              -- 10
end sobel;                                               -- 11
```

Figure 6: An Apply implementation of thresholded Sobel edge detection

Line 1 defines the input, output and constant parameters to the function. The input parameter **inimg** is a window of the input image. The constant parameter **thresh** is a threshold. Edges which are weaker than this threshold are suppressed in the output magnitude image, **mag**. Line 3 defines **horiz** and **vert** which are internal variables used to hold the results of the horizontal and vertical Sobel edge operator.

Line 1 also defines the input image window. It is a 3×3 window centered about the current pixel processing position, which is filled with the value 0 when the window lies outside the image. This same line declares the constant and output parameters to be floating-point scalar variables.

The computation of the Sobel convolutions is implemented by the straight-forward expressions on lines 5 through 7. These expressions are readily seen to be a direct implementation of the convolutions in Figure 5.

5.3.3 Border Handling

Border handling is always a difficult and messy process in programming kernel operations such as Sobel edge detection. In practice, this is usually left up to the programmer, with varying results – sometimes borders are handled in one way, sometimes another. Apply provides a uniform way of resolving the difficulty. It supports border handling by extending the input images with a constant value. The constant value is specified as an assignment. Line 1 of Figure 6 indicates that the input image `inimg` is to be extended by filling with the constant value 0.

If the programmer does not specify how an input variable is to be extended as the window crosses the edge of the input image, Apply handles this case by not calculating the corresponding output pixel.

We plan to extend the Apply language with two other methods of border handling: extending the input image by replicating border pixels, and allowing the programmer to write a special-purpose routine for handling border pixels.

5.4 Apply on Warp

The implementation of Apply on Warp employs straight-forward raster processing of the images, with the processing divided among the cells as described in Section 5.2. The Sobel implementation in Figure 6 processes a 512×512 image on a 10 cell Warp in 330 ms, including the I/O time for the Warp machine.

5.5 Apply on Uni-processor Machines

The same Apply compiler that generates Warp code also can generate C code to be run under UNIX. We have found that an Apply implementation is usually at least as efficient as any alternative implementation on the same machine. This efficiency results from the expert knowledge which is built into the Apply implementation but which is too verbose for the programmer to work with explicitly. (For example, Apply uses pointers to move the operator across the image, instead of moving data). In addition, Apply focuses the programmer's attention on the details of his computation, which often results in improved design of the basic computation.

The Apply implementation for uni-processor machines relies upon a subroutine library which was previously developed for this purpose. The routines are designed to efficiently pass a processing kernel over an image. They employ data buffering which allows the kernel to be shifted and scrolled over the buffer with a low constant cost, independent of the size of the kernel. The Sobel implementation in Figure 6 processes a 512×512 image on a Vax 11/785 in 30 seconds.

5.6 Apply on Other Machines

Here we briefly outline how Apply could be implemented on other parallel machine types, specifically bit-serial processor arrays, and distributed memory general purpose processor machines. These two types of parallel machines are very common; many parallel architectures include them as a subset, or can simulate them efficiently.

5.6.1 Apply on Bit-serial Processor Arrays

Bit-serial processor arrays [8] include a great many parallel machines. They are arrays of large numbers of very simple processors which are able to perform a single bit operation in every machine cycle. We assume only that it is possible to load images into the array such that each processor can be assigned to a single pixel of the input image, and that different processors can exchange information locally, that is, processors for adjacent pixels can exchange information efficiently. Specific machines may also have other features that may make Apply more efficient than the implementation outlined here.

In this implementation of Apply, each processor computes the result of one pixel window. Because there may be more pixels than processors, we allow a single processor to implement the action of several different processors over a period of time, that is, we adopt the Connection Machine's idea of *virtual processors* [17].

The Apply program works as follows:

- Initialize: For $n \times n$ image processing, use a virtual processor network of $n \times n$ virtual processors.
- Input: For each variable of type `IN`, send a pixel to the corresponding virtual processor.
- Constant: *Broadcast* all variables of type `CONST` to all virtual processors.
- Window: For each `IN` variable, with a window size of $m \times m$, shift it in a spiral, first one step to the right, then one step up, then two steps two the left, then two steps down, and so on, storing the pixel value in each virtual processor the pixel encounters, until a $m \times m$ square around each virtual processor is filled. This will take m^2 steps.
- Compute: Each virtual processor now has all the inputs it needs to calculate the output pixels. Perform this computation in parallel on all processors.

Because memory on these machines is often limited, it may be best to combine the "window" and "compute" steps above, to avoid the memory cost of prestoring all window elements on each virtual processor.

5.6.2 Apply on Distributed Memory General Purpose Machines

Machines in this class consist of a moderate number of general purpose processors, each with its own memory. Many general-purpose parallel architectures implement this model, such as the Intel iPSC [19] or the Cosmic Cube [31]. Other parallel architectures, such as the shared-memory BBN Butterfly [9, 26], can efficiently implement Apply in this way; treating them as distributed memory machines avoids problems with contention for memory.

This implementation of Apply works as follows:

- Input: If there are n processors in use, divide the image into n regions, and store one region in each of the n processors' memories. The actual shape of the regions can vary with the particular machine in use. Note that compact regions have smaller borders than long, thin regions, so that the next step will be more efficient if the regions are compact.
- Window: For each `IN` variable, processors exchange rows and columns of their image with processors holding an adjacent region from the image so that each processor has enough of the image to compute the corresponding output region.
- Compute: Each processor now has enough data to compute the output region. It does so, iterating over all pixels in its output region.

5.6.3 Apply on the Hughes HBA

Apply has been implemented on the Hughes HBA computer by Richard Wallace of Carnegie Mellon and Hughes. In this computer, several MC68000 processors are connected on a high-speed video bus, with an interface between each processor and the bus that allows it to select a subwindow of the image to be stored into its memory. The input image is sent over the bus and windows are stored in each processor automatically using DMA. A similar interface exists for outputing the image from each processor. This allows flexible real-time image processing.

The Hughes HBA Apply implementation is straightforward and similar to the Warp implementation. The image is divided in "swaths," which are adjacent sets of rows, and each processor takes one swath. (In the Warp implementation, the swaths are adjacent sets of columns, instead of rows). Swaths overlap to allow each processor to compute on a window around each pixel. The processors independently compute the result for each swath, which is fed back onto the video bus for display.

5.7 Summary

We have described our programming techniques for low-level vision on Warp. These techniques began with simple row-by-row image processing macros, which are still in use for certain kinds of algorithms, and led to the development of Apply, which is a specialized programming language for low-level vision on Warp.

We have defined the Apply language as it is currently implemented, and described its use in low-level vision programming. Apply is in daily use at Carnegie Mellon for Warp and vision programming in general; it has proved to be a useful tool for programming under UNIX, as well as an introductory tool for Warp programming.

The Apply language crystallizes our ideas on low-level vision programming on Warp. It allows the programmer to treat certain messy conditions, such as border conditions, uniformly. It also allows the programmer to get consistently good efficiency in low-level vision programming, by incorporating expert knowledge about how to implement such operators.

One of the most exciting characteristics of Apply is that it may be possible to implement it on diverse parallel machines. We have outlined such implementations on bit-serial processor arrays and distributed memory machines. Implementation of Apply on other machines will make porting of low-level vision programs easier, should extend the lifetime of programs for such supercomputers, and will make benchmarking easier.

6 Symmetric Texel Detection on Warp

In this research, repetitive textures are analyzed by using local point symmetry to detect the texture elements. Point symmetry is detected by an Analysis of Variance (ANOVA) [28] statistical test which is applied to a window surrounding each pixel location.

The ANOVA method consists of partitioning the variance of the data into two portions: that which is explained by the model and that which remains unexplained. The method is applied at each pixel location to measure point symmetry. The model assumes that pixels which are located opposite each other should have similar intensities. The variance explained by the model is given by the following equation, in which I represents the window around each pixel, and W is a weighting function used to emphasize particular pixels around the central pixel, for example by using a circular Gaussian weighting function:

$$SS_m = \sum_{ij} W_{ij}\left(\frac{I_{ij}+I_{-i-j}}{2}-\bar{I}\right)^2, \text{ where } \bar{I} = \frac{\sum_{ij} W_{ij} I_{ij}}{\sum_{ij} W_{ij}}.$$

The unexplained residual variance is given by the following equation:

$$SS_r = \sum_{ij} W_{ij}\left(I_{ij}-\frac{I_{ij}+I_{-i-j}}{2}\right)^2 = \frac{1}{4}\sum_{ij} W_{ij}\left(I_{ij}-I_{-i-j}\right)^2.$$

The ratio of these two quantities is an F statistic [28]. However, the simple ratio has two faults: it is very sensitive to noise in low-contrast portions of the image (such as sky), and its values are unbounded. We therefore use the following ratio:

$$S = \frac{SS_m}{SS_r + SS_m + V}.$$

In this equation, V is a constant which is used to suppress the response to noise, and is roughly equal to the noise variance in the image multiplied by the sum of W_{ij}. The ratio S is bounded below by zero and above by one. Local peaks in an image of S values represent points of local symmetry.

In the Warp implementation of this algorithm, a pair of nested loops over the input image window compute the weighted mean surrounding each pixel. A second pair of nested loops compute SS_m and SS_r. This implementation involves 1321 floating-point multiplications and 1982 floating-point additions per pixel. For a 512×512 image, 346 million multiplications are required and 519 million additions. The prototype Warp processes a 512×512 image in 30s. The same processing would take more than an hour on a SUN-3.

7 References

[1] Arnould, E., Kung, H.T., Menzilcioglu, O. and Sarocky, K.
A Systolic Array Computer.
In *Proceedings of 1985 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 232-235. March, 1985.

[2] *Reference Manual for the Ada Programming Language*
MIL-STD 1815 edition, United States Department of Defense, AdaTEC, SIGPLAN Technical Committe on Ada, New York, N.Y. AdaTEC, 1982.
Draft revised MIL-STD 1815. Draft proposed ANSI Standard document.

[3] Aho, A., Hopcroft, J.E. and Ullman, J.D.
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Massachusetts, 1975.

[4] Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A.
Warp Architecture and Implementation.
In *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 346-356. June, 1986.

[5] Annaratone, M., Bitz, F., Deutch, J., Hamey, L., Kung, H. T., Maulik, P., Ribas, H., Tseng, P. and Webb, J.
Applications Experience on Warp.
In *Proceedings of the 1987 National Computer Conference.* AFIPS, 1987.

[6] Annaratone, M., Arnould, E., Cohn, R., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K., Senko, J., and Webb, J.
Warp Architecture: From Prototype to Production.
In *Proceedings of the 1987 National Computer Conference.* AFIPS, 1987.

[7] Batcher, K. E.
Design of a Massively Parallel Processor.
IEEE Transactions on Computing C-29:836-840, 1980.

[8] Batcher, K. E.
Bit-serial parallel processing systems.
IEEE Trans. Computer C-31(5):377-384, May, 1982.

[9] BBN Laboratories.
The Uniform System Approach to Programming the Butterfly Parallel Processor
1 edition, Cambridge, MA, 1985.

[10] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D.
Programming Warp.
In *COMPCON Spring '87*, pages 268-271. IEEE Computer Society, 1987.

[11] Bruegge, B., Chang, C., Cohn, R., Gross, T., Lam, M., Lieu, P., Noaman, A. and Yam, D.
The Warp Programming Environment.
In *Proceedings of the 1987 National Computer Conference.* AFIPS, 1987.

[12] Gross, T. and Lam, M.
Compilation for a High-performance Systolic Array.
In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 27-38. ACM SIGPLAN, June, 1986.

[13] Gross, T., Kung, H.T., Lam, M. and Webb, J.
Warp as a Machine for Low-level Vision.
In *Proceedings of 1985 IEEE International Conference on Robotics and Automation*, pages 790-800. March, 1985.

[14] Hamey, L., H. Printz, D. Reece, and S. Shafer.
A Programmer's Guide to the Generalized Image Library.
1987.

[15] Hamey, L. G. C., Webb, J. A., and I-C. Wu.
Low-level Vision on Warp and the Apply Programming Model.
Parallel Computation and Computers for Artificial Intelligence.
Kluwer Academic Publishers, 1987.
Edited by Janusz Kowalik.

[16] R. M. Haralick.
Digital step edges from zero crossings of second directional derivatives.
IEEE Transactions on Pattern Analysis and Machine Intelligence 6:58-68, 1984.

[17] Hillis, W. D.
The Connection Machine.
The MIT Press, Cambridge, Massachusetts, 1985.

[18] T. S. Huang, G. J. Yang, and G. Y. Tang.
A fast two-dimensional median filtering algorithm.
In *International Conference on Pattern Recognition and Image Processing*, pages 128-130. IEEE, 1978.

[19] *iPSC System Overview*
Intel Corporation, 1985.

[20] Kernighan, B. W. and D. M. Ritchie.
The M4 Macro Processor.
In *Unix Programmer's Manual.* Bell Laboratories, Murray Hill, NJ 07974, 1979.

[21] Kung, H.T. and Menzilcioglu, O.
Warp: A Programmable Systolic Array Processor.
In *Proceedings of SPIE Symposium, Vol. 495, Real-Time Signal Processing VII*, pages 130-136. Society of Photo-Optical Instrumentation Engineers, August, 1984.

[22] Kung, H.T. and Webb, J.A.
Global Operations on the CMU Warp Machine.
In *Proceedings of 1985 AIAA Computers in Aerospace V Conference*, pages 209-218. American Institute of Aeronautics and Astronautics, October, 1985.

[23] Kung, H. T. and Webb, J. A.
Mapping Image Processing Operations onto a Linear Systolic Machine.
Distributed Computing 1(4):246-257, 1986.

[24] Lasser, C.
*The Complete *Lisp Manual*
Thinking Machines Corporation, Cambridge, Massachusetts, 1986.

[25] Little, J. J., Glelloch, G., and Cass, t.
Parallel algorithms for computer vision on the connection machine.
In *Image Understanding Workshop*, pages 628-638. DARPA ISTO, Feb, 1987.

[26] Olson, T. J.
An Image Processing Package for the BBN Butterfly Parallel Processor.
Butterfly Project Report 9, University of Rochester, Department of Computer Science, August, 1985.

[27] Preparata, F.P. and Shamos, M.I.
Computational Geometry: In Introduction.
Springer-Verlag, New York, 1985.

[28] Rao, C. R.
Linear Statistical Inference and Its Applications.
Wiley, 1973.
Second Edition.

[29] Rosenfeld, A.
A Report on the DARPA Image Understanding Architectures Workshop.
In *Image Understanding Workshop*, pages 298-301. Defense Advanced Research Projects Agency, Los Angeles, California, February, 1987.

[30] Schimmel, D.E. and Luk, F.T.
A New Systolic Array for the Singular Value Decomposition.
In *Proceedings of 1986 Conference on Advanced Research in VLSI*, pages 205-217. M.I.T., April, 1986.

[31] Seitz, C.
The Cosmic Cube.
Communications of the ACM 28(1):22-33, January, 1985.

[32] Electrotechnical Laboratory.
SPIDER (Subroutine Package for Image Data Enhancement and Recognition).
Joint System Development Corp., Tokyo, Japan, 1983.

[33] Thinking Machines Corporation.
Connection Machine Model CM-2 Technical Summary.
HA 87-4, Thinking Machines Corporation, April, 1987.

[34] Waltz, D. L.
Applications of the Connection Machine.
IEEE Computer 20(1):85-97, January, 1987.